Interaction Between Stampede Runtime and Operating Systems

Arnab Paul, Nissim Harel, Sameer Adhikari, Bikash Agarwalla, Umakishore Ramachandran, Ken Mackenzie
College Of Computing, Georgia Institute of Technology
801 Atlantic Drive, NW, Atlanta, GA 30332-0280, USA
rama@cc.gatech.edu

Abstract

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements, making them good candidates for executing on parallel architectures such as SMPs and clusters of SMPs. The necessity to reason about program behavior along the time dimension is an important characteristic of these types of applications. *Stampede* is a cluster programming system that is designed to meet many of the challenges in such applications. Stampede supports time-sequenced data items, and thus facilitates temporally correlating data items from different streams. The system performs automatic garbage collection of data items no longer needed by any application thread. The Stampede system has been built as a runtime library on top of standard operating systems. In this paper, we study the interaction between the Stampede runtime system and the underlying operating system. The study is conducted on two identical hardware platforms running Solaris and Linux, respectively. A cycle accurate event logging facility using the CPU cycle counter is at the core of this study. There are several interesting insights coming from this study. First, memory allocation does not take up a significant amount of the execution time despite the interactive and dynamic nature of the application domain. Second, the Stampede runtime does not pose a significant overhead over raw messaging for structuring such applications. Third, the results suggest that the thread scheduler on Linux may be more responsive than the one on Solaris. Fourth, the messaging layer spends quite a bit of time in synchronization operations.

1 Introduction

Emerging application domains such as interactive vision, animation, and multimedia collaboration display dynamic scalable parallelism, and high computational requirements, making them good candidates for executing on parallel architectures such as SMPs and clusters of SMPs. There are some aspects of these applications that set them apart from scientific applications that have been the main target of high performance parallel computing in recent years. First, time is an important attribute in such emerging applications due to their interactive nature. In particular, they require the efficient management of temporally evolving data. For example, a stereo module in an interactive vision application may require images with corresponding timestamps from multiple cameras to compute its output, or a gesture recognition module may need to analyze a sliding window over a video stream. Second, both the data structures as well as the producer-consumer relationships in such applications are dynamic and unpredictable at compile time.

To simplify the development of such applications, we have designed a programming system called *Stampede* [8] that offers many of the needed functionalities (such as high-level data sharing abstractions, dynamic cluster-wide threads, support for data parallelism, and multiple address spaces). Stampede provides a novel cluster-wide data structure called *Space-Time Memory* (STM) [11] to enable interactive multimedia applications to manage a collection of time-sequenced data items simply, efficiently, and transparently across a cluster. STM isolates the application programmer from low level details by providing a high level interface that subsumes buffer management, inter-thread synchronization, and location transparency for data produced and accessed anywhere in the cluster. STM also automatically handles garbage collection of data items that will no longer be accessed by any of the application threads.

Stampede encompasses key design features for supporting interactive multimedia applications on clusters. It is implemented as a runtime library on top of standard operating systems and standard messaging layers. The execution time of a Stampede application can be divided into four parts: (a) application logic, (b) blocked for resources (c) Stampede runtime, and (d) messaging layer. Quantifying the amount of time spent by typical Stampede applications into these four parts can yield valuable insights on both the applications themselves as well as on the characteristics of complex runtime libraries such as Stampede. Further, teasing out the times spent in dynamic memory allocation (such as mallocs and frees), and synchronization operations (such as locks and unlocks) for the runtime and messaging layers allows us to understand the relationship of the Stampede system to the underlying operating system.

Further, Stampede is available on several different cluster platforms including x86-Solaris, x86-Linux, and Alpha-Tru64. Thus studying the interaction of its runtime behavior with respect to the amount of time on various platforms can lead to new insights on the way resources (memory, processor, and network) are managed in standard operating systems. To make the comparison meaningful, we have restricted this study to two identical hardware platforms, one running Solaris and the other running Linux. To perform such a study, we have implemented an elaborate measurement infrastructure that produces cycle accurate timing for any code block of interest.

The key contributions of this paper are:

- an infrastructure for accurate timing measurements,
- comparison of the execution time breakdown of two novel interactive streaming video applications into application logic, blocked for resources, Stampede runtime, and messaging time on two identical hardware platforms running two different flavors of Unices, and
- quantification of times spent for dynamic memory allocation and synchronization operations by the runtime.

We describe the Stampede programming system in Sec. 2. The questions being investigated in this paper is discussed in Sec. 3. The measurement infrastructure we have implemented in Stampede is described in Sec. 4. We discuss the performance results in Sec. 5. Related work is discussed in Sec. 6. Concluding remarks are presented in Sec. 7.

2 Stampede Programming System

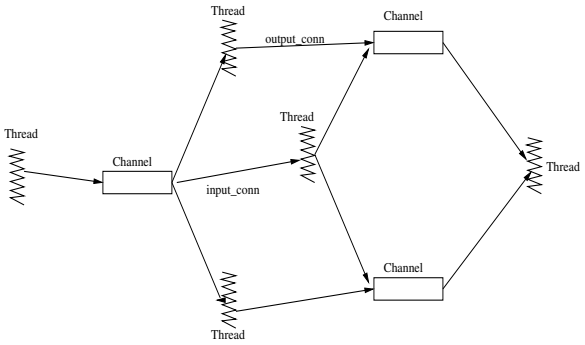

Figure 1: Computational Model: Dynamic Thread Channel Graph

The computational model supported by Stampede is shown by the thread-channel graph in Figure 1. The threads can run any node of the cluster and the channels serve as the application level conduits for time-sequenced stream data among the threads.

2.1 Architecture

The Stampede architecture has the following main components: *Channels*, *Queues*, *Threads*, *Garbage Collection*, *Handler Functions*, and *Real-time Guarantees*.

Threads, Channels, and Queues: Stampede provides a set of abstractions across the entire cluster: threads, channels, and queues. Stampede threads can be created in different protection domains (address spaces) [8]. Channels (queues) are cluster-wide unique names, and serve as containers for *time-sequenced* data. They facilitate inter-thread communication and synchronization regardless of the physical location of the threads and channels (queues). A thread (dynamically) connects to a channel (queue) for *input* and/or *output*. Once connected, a thread can do I/O (get/put items) on the channel (queue). The items represent some application-defined stream data (e.g. video frames). The timestamps associated with an item in a channel (queue) is user defined (e.g. frame numbers in a video sequence). The collection of time-sequenced data in the channels and queues is referred to as *space-time memory* (STM) [11]. At the application level, the Stampede threads perform I/O on the channels and queues, and do not deal with any low level synchronization (such as locks or barriers) among themselves. The put/get operations implicitly combine synchronization with data transfer. These operations are synchronous and atomic with respect to the specific channel. The operations themselves can be blocking or non-blocking. For e.g., a *blocking get* on a channel for a particular timestamped item (the timestamp can be a wildcard such as "latest" and "earliest") returns when the item is actually available, blocking the calling thread if necessary until the item arrives on the channel. A *non-blocking get* returns immediately with or without the requested item (the return code indicates success or failure of the operation).

Conceptually a Stampede computation with threads, channels, and queues is akin to a distributed set of processes connected by sockets. The power of Stampede is the ability to reason about program behavior based on time, and temporal correlation among data generated by different sources.

Garbage Collection: API calls in Stampede facilitate a given thread to indicate that an item (or a set of items) in a channel or queue is garbage so far as it is concerned. Using this per-thread knowledge, Stampede automatically performs distributed garbage collection [7] of timestamps (i.e. items with such timestamps) that are of no interest to any thread in the Stampede computation.

Handler Functions: Stampede allows association of *handler functions* with channels (queues) for applying a user-defined function on an item in a channel (queue). The handler functions are handy in various situations. To transport a complex data structure on channels, these functions can define the serialization (de-serialization) used by Stampede. Similarly Stampede can invoke a handler registered by a thread, when an item becomes garbage.

Real-time Guarantees: The timestamp associated with an item is an indexing system for data items. For pacing a thread relative to real time, Stampede provides an API borrowed from the Beehive system [13]. Essentially, a thread can declare real time interval at which it will re-synchronize with real time, along with a tolerance and an exception handler. As the thread executes, after each "tick", it performs a Stampede call attempting to synchronize with real time. If it is early, the thread is blocked until that synchrony is achieved. It if is late by more than the specified tolerance, Stampede calls the thread's registered exception handler which can attempt recovery in an application specific manner.

2.2 Implementation

Stampede is implemented as a C runtime library on top of several clustered SMP platforms including DEC Alpha-Digital Unix 4.0 (Compaq Tru64 Unix), x86-Linux, x86-Solaris, and x86-NT. It uses a message-passing substrate called CLF, a low level packet transport layer developed originally at Digital Equipment Corporation's Cambridge Research Lab (which has since become Compaq/HP CRL). CLF provides reliable, ordered, point-to-point packet transport between Stampede address spaces, with the illusion of an infinite packet queue. It exploits shared memory within an SMP, and any available cluster interconnect between the SMPs, including Digital Memory Channel [5], Myrinet [3] (using the Myricom GM library), and Gigabit Ethernet (using UDP).

The Stampede runtime is interesting since it implements the high level computational abstractions and the distributed garbage collection. There are several points of interaction between the Stampede runtime and the

underlying operating system:

1. Stampede threads are implemented using the threading facility in the operating system (such as pthreads in Unices). Thus the scheduling of Stampede threads is entirely under the control of the native operating system thread scheduler.

2. The implementation of the Stampede API (such as put/get on channels) uses low level synchronization mechanisms (such as pthread-lock in Unices) provided by the operating system for ensuring the desired semantics of these APIs.

3. The Stampede implementation uses the dynamic memory allocation mechanisms in the operating system (such as malloc and free).

4. Reliable messaging via CLF is implemented on top of the messaging stack of the operating system (such as UDP in Unices).

3 Scope of this Paper

The primary focus of this paper is to quantitatively present the interaction between the Stampede runtime and the underlying operating system. In particular, we address the following questions:

1. For a given execution of a Stampede application, how much time is spent in each of *application logic*, *blocked for resources*, *Stampede runtime*, and *messaging layer*? The first component is the actual work done by the application. The second component is the wait time incurred in the application possibly due to work imbalance in the structure of the application (leading to gets before puts, and extra remote communications). The third is the time incurred for traversal of the Stampede API code path, while the fourth is the time incurred in messaging when the target of the get/put is a remote cluster node. The difference between the last two components quantifies the overhead for using the Stampede high level APIs for structuring such cluster applications over raw messaging. Such a breakdown can yield valuable insights on both the applications themselves as well as on the characteristics of complex runtime libraries such as Stampede.

2. For a given execution of a Stampede application, how much time is spent in *dynamic memory allocation*, and *synchronization operations* by the Stampede runtime and the messaging layers? Such a breakdown will help us understand the relationship of the Stampede system to the underlying operating system.

3. What impact do different operating systems have on the above two items? Studying the interaction of Stampede runtime with different operating systems can lead to new insights on the way resources (memory, processor, and network) are managed in standard operating systems. To make the comparison meaningful, we have restricted this study to two identical hardware platforms, one running Solaris 7 for x86, and the other running RedHat Linux 7.1. The hardware configuration for each set up is a four node cluster interconnected by 100Mbps Fast Ethernet, with each node consisting of Dual Pentium II 300MHz processors with 512MB RAM and 4GB SCSI disk. The CLF messaging layer uses UDP over Fast Ethernet.

   Just as a point of reference for the above comparison, we also show the performance of Stampede on a more state-of-the-art cluster interconnected by 1.2 Gigabit Ethernet, in which each node consists of 8-way 550 MHz Pentium III Xeon processors with 4GB RAM and 18GB SCSI disks, and running RedHat Linux 7.1.

4 Measurement Infrastructure

To perform such a study, we have developed an elaborate measurement infrastructure in Stampede. This infrastructure consists of two parts: *event logging*, and *postmortem analysis*.

4.1 Event Logging

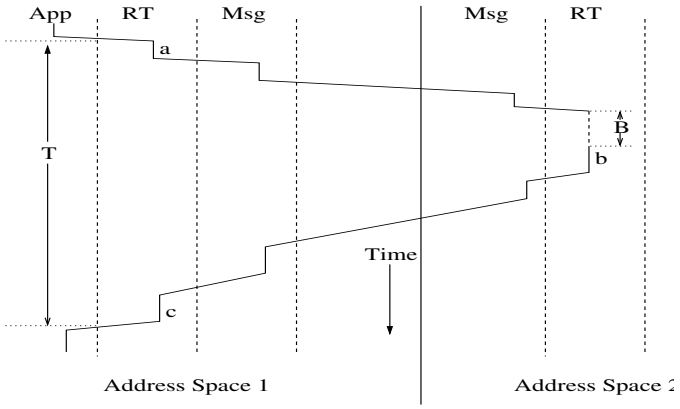

Figure 2: Timeline for a remote operation spanning two address spaces. RT - Stampede Runtime, Msg - Messaging Layer, App - Application logic.

The intent is to be able to accurately accumulate the time spent in any and every piece of code segment for the purposes of answering the questions we raised in the previous section. To this end we have implemented an event logging mechanism. The basic idea is to declare *events* that are names meaningful to the code that is being timed. For e.g., if we want to accumulate the time spent in a procedure body, we place a call to the timing function at the entry (start time) and exit (end time) points. The event logging mechanism records the start and end time along with a mnemonic user-specified event name that uniquely identifies the code fragment that is being timed. The events we wish to measure are very short (as small as few tens of instructions). Further we wish to measure a large number of events (on the order of a million for an application run). Thus the time to record each event should be small such that total time to record these events does not significantly affect the performance of the original application. Therefore, the standard system supported timing calls (such as the Unix gettimeofday) are inappropriate for this kind of event logging since they are quite expensive per call, and do not offer the fine granularity that is required.

Fortunately, most modern processors such as the Intel Pentium line have a CPU cycle counter that is accessible as a program readable register. Our event logging mechanism uses this cycle counter for recording the time. The log is maintained as a buffer in main memory, and each log entry consists of the unique event name, and start and end times for that event. The time for recording the log for an event (a few memory write instructions) is assumed to be small in comparison to the code fragments that need to be timed. Further, we experimentally verified that recording these events themselves does not significantly perturb the overall execution of the original application. The logging subsystem maintains two in-memory buffers. When one in-memory buffer is approaching fullness, the logging subsystem switches to the other buffer, and (via DMA) flushes out the first buffer to the disk in binary form. While this flushing does not use processor cycles, it could perturb the normal application execution since there could be contention for the memory bandwidth. In order to keep this perturbation to a minimum, we pick a large enough buffer size so that the number of disk I/Os during the application execution is kept to a minimum. Typically event log records are about 16 bytes. With a 4MB buffer size, we can ensure that

there are only 4 disk I/Os to generate a log file of a million events.

There are three interesting situations that have to be handled. First, on an SMP the start and end times for a given event could be recorded from different processors due to the vagaries of thread scheduling by the underlying operating system. Fortunately, the CPU cycle counters are initialized at boot time by the operating system and thus remain in sync modulo clock drift on the processors. Experimentally we verified that the cycle counters are in sync and off by at most a few ticks which does not affect the validity of the events being timed which are usually several hundred cycles. Second, the counters should be big enough. The x86 cycle counters are 64 bits in length so there is no worry of the counter wrapping around. Third, an event of interest may cross machine boundary (for e.g. a remote get operation). In this case, such a "macro event" has to be broken up into smaller events, each of which can be locally timed.

4.2 Postmortem Analysis

A postmortem analysis program reads in the events from the log file on the disk and computes the times for macro events of interest. For e.g. Figure 2 shows an operation (such as a remote get) that spans 2 address spaces. The request for an item is generated in address space 1, and is actually fulfilled in address space 2. As can be seen from the figure, if the macro event of interest is the Stampede runtime overhead for this operation then it is $a + b + c$, where a and c are events recorded in address space 1, and b is an event recorded in address space 2.

5 Performance Results

As we mentioned in Sec. 3, we use two identical hardware setup one running Solaris and the other running Linux for this study. We perform two sets of measurements. The first is a set of microbenchmarks that compares the thread scheduling performance, and the message throughput at CLF (using UDP) and Stampede levels on the two platforms. The second set is at the level of Stampede applications. The microbenchmarks are useful for explaining some of the application level results.

5.1 Microbenchmarks

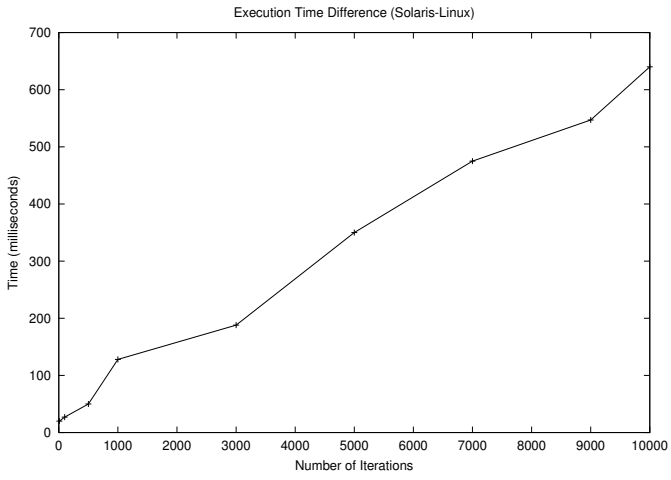

Figure 3: Difference in execution time between Solaris and linux due to scheduling

Thread Scheduling Performance. We perform an experiment to compare responsiveness of the thread scheduler on Linux and Solaris. Recall that the hardware platform consists of dual processor nodes. Three threads are spawned on one node. Two of them keep computing continuously. The third thread alternates between sleeping for a while and computing for a while (quantified as number of iterations in a loop). We consistently found that the total execution time is more on Solaris and the disparity steadily increases with the number of iterations of the third thread. Since the hardware is identical, the most likely reason for this discrepancy is that when the third thread goes to sleep, the vacated processor is not immediately assigned to one of the other two compute threads. In Figure 3 we plot the *difference* between the execution times on Solaris and Linux, respectively. The difference starts at about 20ms for 10 iterations and gets as large as 600ms for 10000 iterations.

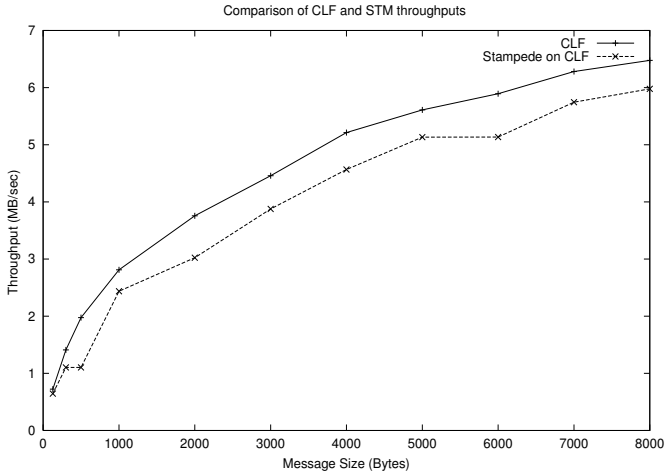

Figure 4: Comparison of CLF and Stampede throughput on Linux

Messaging Performance. We found the performance of CLF messaging to be almost the same on both platforms using a ping-pong test. Figure 4 shows the throughput for Stampede remote *put* operation for different data sizes on Linux compared to the throughput obtained for CLF ping-pong test for the same message size. As can be seen the throughput at the Stampede level is not significantly lower than that at the CLF level, establishing the fact that there is not a significant overhead for using the higher level abstractions of Stampede compared to raw messaging. We obtained similar results for Solaris.

5.2 Application Level Performance

We consider two applications.

1. Mixing of Motion Detector Outputs. Figure 5 shows the task data pipeline of this application, which basically mixes video inputs from a number of clients and sends them back out to each client. A client is composed of a producer-consumer pair. Each producer thread captures images from a camera connected to it and outputs them to a channel. A motion detector thread, dedicated to this producer, picks up images from the associated channel and computes a blob of motion in this image and outputs the result to another channel. A mixer thread picks up corresponding blob outputs from different detectors, combines them and puts the composite onto an output channel from where it is shipped to multiple consumers when they perform get operations. We experimented with different configurations of this application. Each configuration is characterized by the number of clients (producer-consumer pairs) number of address spaces the application is spread across and how the threads are placed in those address spaces.

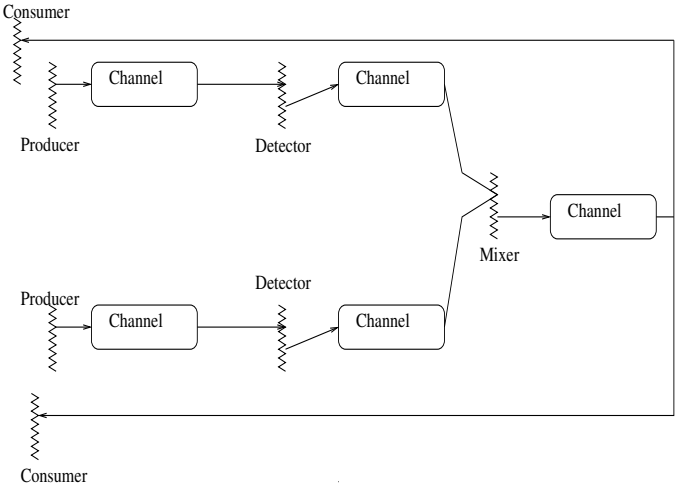

Figure 5: Mixing of Motion detector outputs.

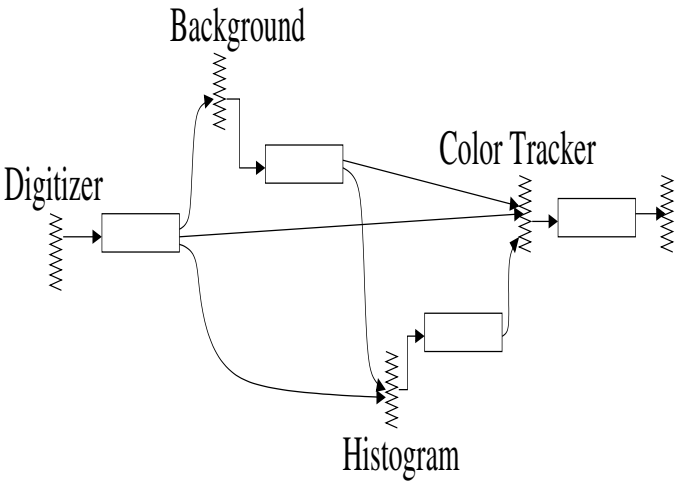

Figure 6: Color Model based Tracker

2. Color Model based Tracker. Figure 6 shows the task data pipeline of the color model based tracker. Every thread in this application has an associated output channel. A digitizer thread gets input from a camera, and produces a digitized image.The background thread uses the digitized images and does a frame by frame background subtraction. A Histogram thread creates a color histogram of the background subtracted image. The tracker uses the histogram output to look for an object given its color model. The tracker output can be optionally sent to some further stages of analysis.

Both these application rely on the timestamp of the items they obtain from the different channels for temporal correlation.

Overall Performance. The overall performance of both the applications on the two platforms (x86-Solaris and x86-Linux) are roughly the same. The motion detector application gives a frame rate of 14fps for one client (image size 70 KB), while the color tracker gives a frame rate of 13fps (image size 75 KB).

Just as a point of reference we ran the two applications on a state-of-the-art cluster with a Gigabit Ethernet connection which we alluded to in Sec. 3. The motion detector application gives a frame rate of 36fps for one client (image size 70 KB), while the color tracker gives a frame rate of 20fps (image size 75 KB). These results show that these applications are well suited to clusters and that Stampede abstractions support these applications

quite well.

5.3 Breakdown of Execution Time

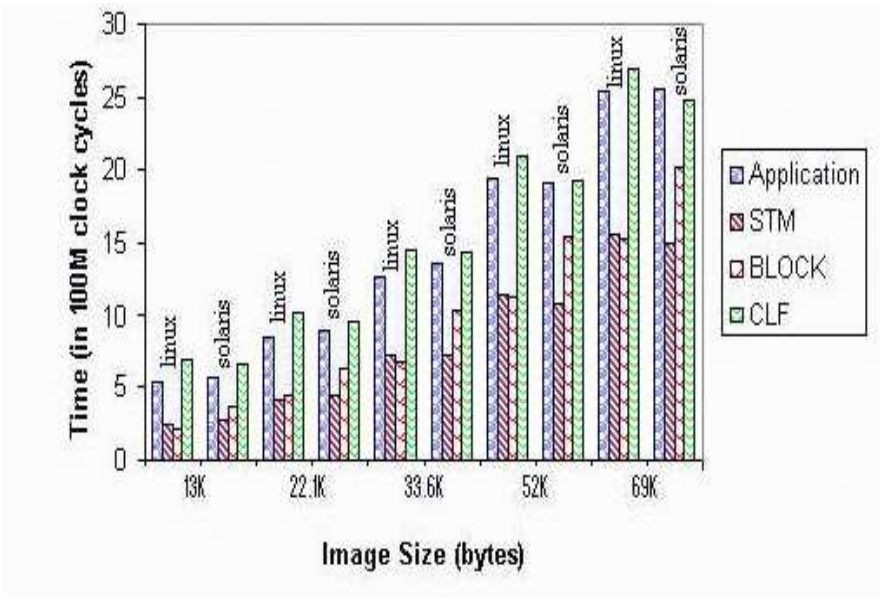

Figure 7: Motion Detector: Execution time split into different layers

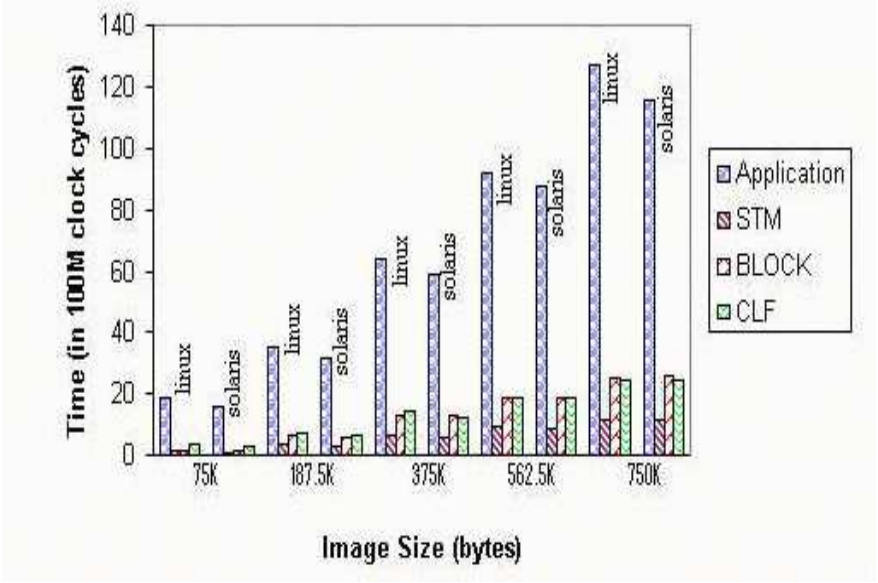

Figure 8: Color Tracker: Execution time split into different layers

As detailed in Sec. 3, we now examine the breakdown of the execution time into the different components such as application logic, blocking, Stampede runtime overhead, and messaging for the two platforms. For the motion detector we use a single client (producer-consumer pair). The image size is varied from 13 KB to 69 KB. The client (producer-consumer) pair is on one address space, the mixer on a second address space, and the detector on a third address space. The channels are created in the address space of the thread that puts items into the channel. For the color tracker the image size is varied from 75 KB to 750 KB. Every thread in the color

tracker application is in an address space by itself. Once again the channels are co-located with the thread that puts items into the channel. For all the experiments, each address space is on a different node of the cluster.

Messaging and Scheduling Comparison. Figure 7 shows the breakdown of execution time for the motion detector. For each image size, respective breakdown on Solaris and Linux are juxtaposed. Figure 8 shows similar breakdown for the color tracker. From Figure 7 it can be seen that the messaging time is always a little lower in Solaris. However, although the application logic and Stampede overhead (labeled STM in the graph) are almost same on both platforms, the blocking time is always significantly more on Solaris than in Linux. This is counter-intuitive. Blocking in the Stampede pipeline is due to waiting for an item in a channel (on a get), or waiting for space on a channel (on a put). The faster the messaging (assuming a constant time in the application or STM), faster the items will come in and be taken away from the channels and one would therefore expect less blocking time. Our experimental results however show otherwise. Results in Figure 8 are also along the same line, though the effect is less prominent because application work is much more compared to other components.

We ascribe this anomaly to the scheduling policy of the operating system. When a thread blocks waiting for an item to become available, it is context switched with other active threads. Blocking is minimized if the thread is scheduled as soon as the desired item becomes available (or taken away). Thus the responsiveness of the scheduler is a determinant for the observed blocking time. A sub-optimal scheduling policy may not schedule the thread at right time and thereby adding more cycles to the blocking event of the same thread. This conjecture is strengthened by the microbenchmark result in Sec. 5.1 which shows that the scheduling on Linux performs better.

Time Spent in Application Logic. From Figure 8, it is interesting to note that the time spent in the Application Logic for the color tracker is always slightly higher for Linux compared to Solaris. Since the hardware platforms are identical and both use the gcc compiler, we ascribe this to implementation differences in the compiler support libraries.

Stampede Runtime Overhead. In Figures 7 and 8, the bars labeled STM is the Stampede runtime overhead. Table 1 shows the percentage of time spent in the Stampede runtime layer for two applications. Experiments are marked from 1 to 5. Which in the case of motion detector means, image sizes 13, 22, 33.6, 52 and 69 Kilobytes and for the color tracker, 75, 187, 375, 562.5, and 750 Kilobytes. As can be seen for each application (on both platforms) the Stampede runtime overhead is roughly a fixed percentage of the execution time, showing the scalability of the Stampede runtime system.

Platform	Experiments				
	1	2	3	4	5
Motion-Detector (Linux)	14.77	15.0	17	18	18.7
Motion Detector (Solaris)	14.9	15.31	16.7	16.7	17.5
Color Tracker (Linux)	5.8	6.3	6.4	6.7	6.14
Color Tracker (Solaris)	5.1	6.0	6.6	6.5	6.5

Table 1: Stampede Runtime Overhead: Percentage of the total time spent in STM.

5.4 Memory and Synchronization Breakdown

This subsection answers the other question raised in Sec. 3, namely, how much time is spent by the Stampede runtime and CLF messaging in dynamic memory allocation and synchronization related activities. Clearly these

are points of interaction with the underlying operating system. Due to the dynamic nature of both the applications, and the large sizes of the data being manipulated, it may be expected that a significant amount of time is spent in dynamic memory allocation operations (such as malloc and free). Table 2 shows the percentage of Stampede runtime spent in memory allocation and freeing for the motion detector. It is interesting to observe that this percentage is quite small. The memory activity is at most 3.2% on Linux and at most 6.7% on Solaris. The numbers for the color tracker, which are not presented in this paper are quite similar. On a similar note we investigated the runtime's thread synchronization activity as well. Table 3 summarizes the percentage of time Stampede runtime spends in thread synchronization for the motion detector application. Once again it is interesting to note that despite the nature of these applications, relatively little time is spent in thread synchronization activity. On Solaris the relative time spent in thread synchronization is higher than on Linux.

CLF's interaction with the memory allocator is negligible. CLF does a one time allocation of a sufficiently big memory region and thenceforth does its own buffer management. Table 4 summarizes CLF's thread synchronization time relative to total time spent in CLF. It is seen that a significant percentage of time is spent in thread synchronization. Further, in contrast to what we observed for the Stampede runtime, the time spent in thread synchronization on Linux is higher than on Solaris for CLF messaging. At the time of writing this paper, we do not have a ready explanation for this result[1].

Platform	Experiments				
	1	2	3	4	5
Linux	3.2	3.2	2.4	2.1	2.1
Solaris	6.7	5.1	3.9	3.8	3.3

Table 2: Percentage of Stampede Runtime spent in Memory allocation/free (Motion Detector).

Platform	Experiments				
	1	2	3	4	5
Linux	1.15	1.6	2.4	2.0	2.0
Solaris	6.8	7	5.9	6.2	4.8

Table 3: Percentage of Stampede Runtime spent in Thread Synchronization (Motion Detector).

Platform	Experiments				
	1	2	3	4	5
Linux	18.1	18.7	19.0	19.6	20.3
Solaris	10.9	10.9	11.5	12.0	11.4

Table 4: Percentage of CLF Messaging spent in Thread Synchronization (Motion Detector).

6 Related Work

We are not aware of any study that has exactly the same goals as ours of quantifying the interaction between a cluster runtime layer and the operating system. There are a number of studies that have looked at providing measurement support. ProfileMe system [4] uses program counter (PC) sampling to relate the hardware events

[1] We will have explanation for this result via microbenchmarking in time for the conference.

(such as cache misses) to the likely instructions that incurred such events. They designed hardware support that allows accruing accurate profile information from out-of-order processors. Continuous profiling [2] is a system that periodically samples the PC on an Alpha SMP, and stores all the hardware counter values at the time of sampling. Periodically these images are written out to disk. Their system uses 1 to 3% of CPU time, modest amount of memory and disk. Profile information is gathered for the entire system allowing them to pinpoint sources of performance problems. Ofelt and Hennessy [9] describe a technique that involves an instrumentation phase followed by an analysis phase to predict the performance of modern processors. Though their goal is different there is some similarity in our measurement infrastructure to their approach. Ammons et al. [1] describe a technique that uses hardware performance counters to predict the hot paths in traditional benchmarks. There is a large body of work that deals with performance characterization of aspects of computer system (memory hierarchy, pipeline performance, etc.) from the point of view specific applications (such as multimedia [12]) or specific programming environments (such as Java [6]).

7 Concluding Remarks

Stampede is a cluster programming library with novel features for supporting emerging interactive stream-oriented applications. Given the novelty of both the application domain as well as the Stampede programming system, this research has attempted to quantitatively present the interaction between the Stampede runtime and the underlying operating system. In order to do this, a cycle accurate event logging facility has been implemented using the CPU cycle counter found in most modern processors. In addition to giving a breakdown of execution times for two video streaming applications, this study also investigates how this breakdown compares for Solaris and Linux on identical hardware configurations. Further, the time spent in performing dynamic memory allocation and synchronization operations both in the Stampede runtime as well as the messaging layer is also quantified.

References

[1] G. Ammons, T. Ball, and J. Larus Exploiting Hardware Performance counters with Flow and Context Sensitive Profiling. In *Proc. ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, 1997

[2] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl Continuous profiling: Where have all the cycles gone?. In*Proc. of the 16th ACM Symposium of Operating Systems Principles (SOSP 97)*, pages 1-14, October 1997

[3] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet – a gigabit-per-second local-area network, Draft 16 Nov 1994.

[4] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos ProfileMe: Hardware Support for Instruction-Level Profiling on Out-of-Order Processors. In *Proc. 30th Annual IEEE/ACM Intl. Symp. on Microarchitecture, Research Triangle Park, NC*, pages 292-302, 1997

[5] R. Gillett. MEMORY CHANNEL Network for PCI: An Optimized Cluster Interconnect. *IEEE Micro*, pages 12–18, February 1996.

[6] J. Kim, and Y. Hsu Memory System Behavior of Jave Programs: Methodology and Analysis In *Proc. of the International Conference on Measurements and Modeling of Computer Systems, Santa Clara, CA*, pages 264-274, 2000

[7] R. S. Nikhil and U. Ramachandran. Garbage Collection of Timestamped Data in Stampede. In *Proc.Nineteenth Annual Symposium on Principles of Distributed Computing (PODC 2000), Portland, Oregon*, July 2000.

[8] R. S. Nikhil, U. Ramachandran, J. M. Rehg, R. H. Halstead, Jr., C. F. Joerg, and L. Kontothanassis. Stampede: A programming system for emerging scalable interactive multimedia applications. In *Proc. Eleventh Intl. Wkshp. on Languages and Compilers for Parallel Computing (LCPC 98), Chapel Hill NC*, August 7-9 1998.

[9] D. Ofelt, and J. L. Hennessy Efficient Performance Prediction for Modern Processors. In *Proc. of the international conference on Measurements and modeling of computer systems, Santa Clara, CA*, pages 229-239, 2000

[10] IEEE. Threads standard POSIX 1003.1c-1995 (also ISO/IEC 9945-1:1996), 1996.

[11] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe. Space-Time Memory: A Parallel Programming Abstraction for Interactive Multimedia Applications. In *Proc. Principles and Practice of Parallel Programming (PPoPP'99), Atlanta GA*, May 1999.

[12] S. Sohoni, R. Min, Z. Xu and Y. Hu A Study of Memory System Performance of Multimedia Applications In *Proceedings of the joint International Conference on Measurement and modeling of computer systems, Cambridge, MA*, pages 206-215, 2001

[13] A. Singla, U. Ramachandran and J. Hodgins, *Temporal Notions of Synchronization and Consistency in Beehive*, 9th Annual ACM SPAA, June 97.